Reading #4

Proving Termination and Analyzing the Running Time of a Simple Algorithm with a While Loop

Example Problem and Algorithm

Once again, consider the following computational problem.

Maximal Element of Integer Array

Precondition: An integer array A, with positive length n, is given as input.

Postcondition: The largest element in the set

$$\{A[0], A[1], \ldots, A[n-1]\}$$

is returned as output.

Consider, as well, the following algorithm.

```
integer arrayMax (integer[] A) }
1. if (A.length == 1) {
2.   return A[0]
   } else {
3.   integer i := 0
4.   integer maxSoFar := A[0]
5.   while (i < A.length - 1) {
6.     i := i + 1
7.     if (maxSoFar < A[i]) {
8.       maxSoFar := A[i]
       }
     }
9.   return maxSoFar
   }
}
```

During the previous reading, the ***partial correctness*** of an algorithm for a given computational problem was defined, and it was proved that this algorithm is partially correct.
However, the algorithm has not been proved to be ***correct*** — because it has not been proved that an execution of this algorithm will always eventually end, if it begins with its problems's precondition being satisfied.

Bound Functions for While Loops

Definition 1. A ***bound function for a while loop*** is a an well-defined integer-valued, total function of some of the inputs, variables and global data that are accessed and modified when the loop is executed — specifically, some of the inputs, variables and global data that are defined when the loop is reached — and that satisfies the following additional properties, if the computational problem's precondition was satisfied when execution of the algorithm began:

(a) When the loop body is executed, the value of this function is decreased by *at least* one before the loop's test is checked again (if it is reached again, at all).

(b) If the value of this function is less than or equal to zero and the loop's test is checked then the test fails (ending this execution of the loop).

Note: This is sometimes called a "loop variant."

Claim 2. *The function* $f(\texttt{A}, \texttt{i}) = \texttt{A.length} - \texttt{i} - 1$ *is a bound function for the* `while` *loop in the* `arrayMax` *algorithm.*

Proof.

- Since `A` is an input integer array and `i` is an integer variable, defined and initialized at step 3, this is certainly a well-defined integer-valued total function of some of the inputs, variables and global data that are defined when the `while` loop in the `arrayMax` algorithm is reached.
- When the body of the `while` loop is executed the value of `i` is increased by one (at step 6), and the type and length of `A` are not changed — so that the value of f is decreased by one.
- If the value of f is less than or equal to zero then $\texttt{A.length} - \texttt{i} - 1 \leq 0$, so that $\texttt{i} \geq \texttt{A.length} - 1$ and the loop test (at line 5) would fail when checked.

It follows, by the definition of a "bound function for a `while` loop", that f is a bound function for the `while` loop in the `arrayMax` algorithm, as claimed. □

Proving Correctness

Proving Termination

The following result will be useful when proving that an algorithms terminates (when executed with the precondition for its problem satisfied).

Theorem 3 (Loop Theorem #2)**.** *Consider a* `while` *loop*

```
while (t) do { S }
```

with a loop test `t` *and a loop body* `S`*. Suppose that the following properties are satisfied.*

(a) If the problem's precondition is satisfied when an execution of the algorithm including this loop begins, then every execution of the loop body (that is part of this execution of the algorithm) ends.

(b) A bound function for this `while` *loop exists.*

Then every execution of this `while` *loop, included in an execution of the algorithm beginning with the problem's precondition satisfied, ends.*
Furthermore, the value of the bound function immediately before the beginning of an execution of this loop is an **upper bound** *for the number of times that the loop body is executed before this execution of the loop ends.*

A supplementary document, including a proof of this result, is also available. It is possible to prove that the execution of a `while` loop — that is part of the execution of an algorithm beginning with its problem's precondition satisfied — always halts, simply by confirming that all of the conditions in the above result are satisfied.

Claim 4. *If the algorithm* `arrayMax` *is executed when the precondition for the "Maximal Element of Integer Array" problem is satisfied then this execution of the algorithm eventually terminates.*

Proof. Consider an execution of the `arrayMax` algorithm when the precondition for the "Maximal Element of Integer Array" problem is satisfied. An integer array `A`, with positive length n, has been given as input.

- If $n = 1$ then the test at step 1 is checked and passed, and the execution of the algorithm ends after the execution of step 2, as required to establish the claim.
- If $n \geq 2$ then the test at step 1 fails, so that steps 3 and 4 are reached, and the `while` loop is reached and executed.

  The body of this loop includes only three steps (one of which is not always executed) and certainly terminates whenever it is executed.

  It follows by Claim 2 that this `while` loop has a bound function.

It now follows by Theorem 3 that every execution of this `while` loop, during an execution of the algorithm starting with its problem's precondition satisfied, will eventually end.

The execution of the algorithm will also end, after an execution of step $\#9$.

It follows that every execution of this algorithm — beginning with its problem's precondition satisfied — terminates, as claimed. □

Completing a Proof of Correctness

Claim 5. *If an algorithm, for a given computational problem, is both partially correct and terminates, whenever it is executed when its problem's precondition is initially satisfied, then this algorithm is correct.*

Proof. This is a straightforward consequence of the definitions of "partial correctness", "termination", and "correctness". □

Corollary 6. *The* `arrayMax` *algorithm is correct — that is, it correctly solves the "Maximal Element of Integer Array" problem.*

Proof. This is now a consequence of Claim 5, the partial correctness of this algorithm (established during the previous reading), and Claim 4. □

Bounding Running Time

We will use "analytical techniques" to discover bounds for running times — as well as storage requirements — and then write proofs that the bounds are correct (if this is not obvious) These techniques are the focus of the rest of this reading and all of the next one.

A Simplification: The "Uniform Cost Criterion"

This is complicated by the fact that is not clear how much the "running time" of a statement in a program would be. Indeed, this might not even be well-defined!
The following is, therefore, a *simplification* or *abstraction*— which often allows useful analyses of the running times and storage requirements to be performed.

Definition 7. The ***Uniform Cost Criterion*** is an abstraction that provides definitions for the running times of steps in an algorithm and for the storage requirements for data accessed and modified by an algorithm: Each step in an algorithm is defined to have running time 1, and each value from an elementary data type (such as `integer` or `string`) is defined to have cost one.

Of course, it is not always clear whether an instruction includes only one "step" or several of them. In order to simplify things further it will be assumed, for the rest of these readings, that executions of each of the ***numbered steps*** in algorithms should have running time one.

Bounding the Number of Steps Needed to Execute a Loop

Recall that Theorem 3 asserts that the value of a bound function for a `while` loop, immediately before the execution of the loop begins, is an **upper bound** for the number of executions of the loop body included in this execution of the loop.

- Claim 2 establishes that `A.length` $-$ `i` $- 1 = n -$ `i` $- 1$ is a bound function for the `arrayMax` algorithm. Since `i` is assigned the value 0 at line 3, and the value of `i` is not changed by the execution of the step at line 4, this function has value $n - 1$ immediately before the execution of the `while` loop begins — so there are at most $n - 1$ executions of the body of this `while` loop.

Note that ***for this particular example*** (but not in general)

- The value of the bound function f is decreased by **exactly** one, every time the loop body is executed.
- The loop's test fails **if and only if** the value of f is less than or equal to zero when the test is checked.
- The loop's test **must** be checked, and the test must fail, in order to an execution of this loop to end — there are no `return` statements in the loop body, or other statements that could cause the execution of the loop to end in another way.

Proof Exercise: Modify the proof of Loop Theorem #2 (in the supplemental notes) to prove that an execution of the `while` loop of this algorithm always includes **exactly** $n - 1$ executions of the loop body.
The number of execution of the loop *test* will always be at most one more than the number of executions of the loop body — because the loop test is checked before every execution of the loop body and (unless the use of something like a `break` statement causes the execution of the loop to end, or a `return` statement causes the execution of the algorithm to end) there will be one more execution of the loop test, which fails, in order to end the execution of the loop.

- It follows that there are n executions of the loop test at line 5.

Suppose now that it has been determined that there at most k executions of the body of a `while` loop during an execution of a `while` loop — so that there are at most k executions of the loop test.
For $1 \leq j \leq k + 1$, let $T_{\textit{test}}(\, j \,)$ be the number of steps used for the $j^{\texttt{th}}$ execution of the loop test (setting this to be 0 if the loop test is not executed a $j^{\texttt{th}}$ time).

- In the example being considered, $T_{\textit{test}}(\, j \,) = 1$ for $1 \leq j \leq n$.

Then the **total** cost of *all* executions of the loop's test, in an execution of the loop, is

$$\sum_{j=1}^{k+1} T_{\textit{test}}(\, j \,).$$

- In the example being considered, this is

$$\sum_{j=1}^{n} 1 = n.$$

For $1 \leq j \leq k$, let $T_{\textit{body}}(\, j \,)$ be the number of steps used for the j^{th} execution of the loop body — which is 0 if there are fewer than j executions of the loop body.

- The loop body consists of a sequence of three statements — lines 6–8. The first two are always executed, with the third executed only if the test at line 7 is passed. None of these causes the loop body to exit before the bottom of the loop is reached, and none of these calls other methods — so that $2 \leq T_{\textit{body}}(\, j) \leq 3$ for $1 \leq j \leq \mathtt{n} - 1$, for the example being considered.

Then the **total** cost of *all* of the executions of the loop body is

$$\sum_{j=1}^{k} T_{\textit{body}}(\, j \,).$$

- In the example being considered, this is at least $2n - 2$, and at most $3n - 3$.

Since every step, in an execution of a `while` loop, is *either*

- part of an execution of the loop's test, or
- part of an execution of the loop body,

it follows that the number of steps required for an execution of a `while` loop is

$$\sum_{j=1}^{k+1} T_{\textit{test}}(\, j \,) + \sum_{j=1}^{k} T_{\textit{body}}(\, j \,)$$

- In the example being considered, this is at least $3n - 2$ and at most $4n - 3$.

Bounding the Number of Steps Needed to Execute and Algorithm

If an algorithm has multiple loops then the above process must be applied to each loop — working from the innermost loop out to the outermost loop, for nested loops — and adding in the number of steps that are *outside* the loops after that.

- If the length of `A` is $n = 1$ then the algorithm executes exactly *two* steps, namely, the steps at lines 1 and 2.

- If the length of `A` is $n \geq 2$ then the test at line 1 fails, so that the execution of the algorithm on input `n` includes an execution of the `while` loop.

- Exactly 3 statements (namely, the statements at lines 1, 3, and 4) are executed before the `while` loop is reached.
- By the above analysis, the execution of the `while` loop includes at least $3n - 2$ steps and at most $4n - 3$ steps.
- Exactly one statement (namely, the statement at line 9) is executed after the execution of the loop ends.

It follows that the execution of the algorithm, on an input array `A` with length $n \geq 2$, includes at least $3n + 2$ steps and at most $4n + 1$ steps.

Conclusion: The number of steps executed by the `arrayMax` algorithm, when it is given an input array `A` with positive length n, is at most

$$T(\mathtt{n}) = \begin{cases} 2 & \text{if } n = 1, \\ 4n + 1 & \text{if } n \geq 2. \end{cases}$$

Note: This is a function of the input, `n`, as required here!

A Mistake That Sometimes Gets Made: Confusing the *bound function* with its *initial value* when bounding the cost of the loop — so that the expression that gets reported also mentions the variable `i`. Since `i` is not an input (and its value changes during the execution of the algorithm) this is meaningless...

Bounding Storage Requirements

Note that this algorithm uses only three inputs and variables, namely,

- `A` — an integer array with length n (so that it costs n to store it, because this requires the storage of n integers)
- `i` — an integer variable (so that it costs 1 to store it)
- `maxSoFar` — another integer variable (so that it costs 1 to store it)

The ***uniform cost criterion*** has been used to calculate the storage requirements for each input or variable, above.
Note, as well that the variables `i` and `maxSoFar` are only declared and stored if $n \geq 2$.
It suffices to add the storage requirements for each input and variable to define the storage requirements for the algorithm — so the storage requirements for *this* algorithm are given by the function

$$S(n) = \begin{cases} 1 & \text{if } n = 1, \\ n + 2 & \text{if } n \geq 2. \end{cases}$$

Summations

Notation for Summations

Assumption: You have seen summations of the form

$$\sum_{j=a}^{b} T(\,j\,)$$

where a and b are integers and "$T(\,j\,)$" is some function of j.

- If $a > b$ then $\displaystyle\sum_{j=a}^{b} T(\,j\,) = 0$, because this is an "empty sum."
- $\displaystyle\sum_{j=a}^{a} T(\,j\,) = T(a)$.
- $\displaystyle\sum_{j=a}^{a+1} T(\,j\,) = T(a) + T(a+1)$.
- $\displaystyle\sum_{j=a}^{a+2} T(\,j\,) = T(a) + T(a+1) + T(a+2)$.
- $\displaystyle\sum_{j=a}^{a+3} T(\,j\,) = T(a) + T(a+1) + T(a+2) + T(a+3)$.

... and so on.
Note: There is nothing special about the name j:

$$\sum_{h=a}^{a+3} T(h) = T(a) + T(a+1) + T(a+2) + T(a+3)$$

and

$$\sum_{\textit{discombobulator}=a}^{a+3} T(\textit{discombobulator}) = T(a) + T(a+1) + T(a+2) + T(a+3)$$

as well.
You are possibly *not* so familiar with another "notation for summations" that sometimes gets used as well: If "*condition on* j" is a condition that identifies a set of elements that j might belong to, and a Boolean condition that is satisfied by finitely many elements *of* that set, then

$$\sum_{\textit{condition on } j} T(\,j\,)$$

is the sum that includes $T(h)$ for every element h of the set that satisfies the condition.

- $\sum_{j \in \mathbb{Z} \text{ and } a \leq j \leq b} T(j)$ is just another way to write $\sum_{j=a}^{b} T(j)$.
- $\sum_{\substack{j \in \mathbb{Z}, 1 \leq j \leq 10, \\ \text{and } j \text{ is prime}}} T(j) = T(2) + T(3) + T(5) + T(7)$.

Some Useful Identities

1. If a and b are integers such that $a \leq b + 1$ then

$$\sum_{j=a}^{b} 1 = b - a + 1.$$

In each of the remaining cases, n is an integer such that $n \geq 0$.

2. $\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$.
3. $\sum_{j=1}^{n} j^2 = \frac{n(n+1)(2n+1)}{6}$.
4. $\sum_{j=1}^{n} j^3 = \frac{n^2(n+1)^2}{4}$.

Mathematical induction can often be used to provide identities like the above ones.
Indeed, it is possible that you were introduced to summations, in a discrete mathematics course, in order to introduce problems that you could solve using induction!
Suggested Exercise (if you feel like you need more practice using mathematical induction): Prove that the last three identities, on the previous slide, are all correct.

Other Stuff

How Else Might We Measure Running Time?

We might also ***run the code and time its execution***.

An Advantage of This Approach:

- It does not require simplifying assumptions, like the "uniform cost criterion."

A Problem With This Approach:

- Execution time is influenced by many factors — some of which have nothing, at all, to do with the algorithm used:
  - *Hardware:* How fast is the CPU? How many CPU's are there?
  - *Compiler and System Software:* Which OS is being used?
  - *Simultaneous User Activity:* How many other people are using this system at the same time as you are? What are they using it for?
  - *Choice of Input Data:* Running times can be very different for different inputs — even ones with the same "size".
  - *The Skill of the Programmer(s) Who Wrote the Program*

What Else Might We Measure?

Other things that are important and that are sometimes measured include:

- The space required to store the program, *itself:* This might be important if the program is to be stored on a low-memory device (like a smart card).
- The time required to code — and *maintain* — the program. Programmers must be paid and software development usually has deadlines!